Vietnam J Comput Sci (2014) 1:231–239
DOI 10.1007/s40595-014-0023-8

REGULAR PAPER

A simple distributed reasoning system for the connection calculus

Adam Meissner

Received: 10 December 2013 / Accepted: 7 May 2014 / Published online: 28 May 2014
© The Author(s) 2014. This article is published with open access at Springerlink.com

Abstract We present a simple, distributed reasoning system for the first order logic, which applies a connection calculus as an inference method. The calculus has been proposed by Bibel as a generalization of some other popular approaches, like the tableau calculus or the resolution-based inference. The system is constructed in a lean deduction style and it has been inspired to some extent by a sequential reasoner leanCoP, implemented in Prolog. Our reasoner has a form of a relational program in the Oz language. In this programming model, a computational strategy is a parameter of a program having a form of a special object called a search engine. Therefore, the same program can be run in various ways, particularly in parallel on distributed machines. For this purpose, we use a parallel search engine available on the Mozart platform, which is a programming environment for Oz. We also describe results of experiments for estimating a speedup obtained by the distributed processing.

Keywords Distributed reasoning system · Connection calculus · Lean deduction · Oz language

1 Introduction

The term *reasoning system* or, synonymously, an *inference system* or a *reasoner* denotes a computer program intended for processing knowledge, which is expressed in a formal calculus, e.g. the first order logic (FOL). Programs of this type are successfully used in various fields, e.g. in expert systems, control in manufacturing, action planning in robotics, predictive analytics or natural language processing. Nowadays, one can observe a growing popularity of "intelligent" tools, which apply miscellaneous forms of reasoning. This, in turn, increases a demand for efficient reasoning systems that can be easily adopted to different tasks and relocated among various computational environments. Unfortunately, the efficiency of reasoners is usually obtained by implementing complex optimization techniques. In consequence, the majority of reasoning systems are complicated constructions, difficult to modify and to transfer from one environment to another. Moreover, the complex system architecture often causes scalability problems and increases the probability of error occurrence.

The exception are so-called *lean reasoning systems*, i.e. relatively small programs containing only basic mechanisms, essential for soundness and completeness of an inference process. Well-known, precursory examples of lean reasoners are: PTTP [11], SATCHMO [6] or leanT^AP [2]. Programs of this type obviously can not solve hard inference problems. Nevertheless, this approach yields various benefits. In contrast to complex systems, lean reasoners are not hard to verify. Moreover, they can be easily modified and adapted to particular applications [1]. They are also remarkably efficient in solving less difficult problems. It follows from a lower overhead for handling their internal parts than it happens for advanced systems. Furthermore, lean reasoners can act as convenient test-beds for comparison of various inference techniques, where the absolute efficiency is not as important as the relative one.

We consider that the computational power of lean reasoning systems can be enhanced by introducing parallel and distributed computations to them. Therefore, we propose a distributed lean reasoner for FOL, as the main contribution of this paper. The key difficulty in designing a program of this type is to preserve its simplicity. In our approach, the rea-

A. Meissner (✉)
Institute of Control and Information Engineering, Poznań University of Technology, pl. M. Skłodowskiej-Curie 5, 60-965 Poznan, Poland
e-mail: Adam.Meissner@put.poznan.pl

soner is constructed in a declarative programming paradigm, which separates a program from a computational strategy. Hence, the reasoning system can keep its small size since it comprises only a description of basic inference instruments, while the operational semantics is defined outside.

Putting this idea into practice, we decided to use the *relational programming* model available in the Oz language [9], where a computational strategy is a parameter of the program execution. Some details of this conception are given in Sect. 4. Furthermore, we chose the *connection calculus* [3] as a reasoning formalism. It was proposed by Bibel as a generalization of some other popular approaches, like the tableau calculus or the resolution-based inference. The idea of parallel reasoning in the connection calculus is not new—it was considered e.g. by the authors of the paper [7] who defined a variant of the calculus suggesting how some inference steps can be parallelized. The authors also implemented their ideas as a short Prolog program, which can be regarded as a lean reasoning system. However, it is not clear if any attempts were made to run this program in parallel. Another successful example of a sequential lean reasoner for the connection calculus, also implemented in the Prolog language, is the system leanCoP [8]. This system is the inspiration for the solution presented here.

The rest of the paper is organized as follows: In Sect. 2 we present the principles of the connection calculus. Section 3 contains a description of the general reasoning algorithm. In Sect. 4 we describe crucial elements of the reasoning system. The description is preceded by the sketch of the relational programming model and distributed computing in the Oz language. Section 5 provides the results of experiments intended for estimating the speedup obtained by distributed computations. Section 6 concludes the paper with some final remarks.

2 Principles of the connection calculus

We briefly present the fundamentals of the connection calculus. In this description we use a standard FOL syntax and semantics as well as some elements of logic programming theory [5]. In particular, the alphabet encompasses constants a, b, c, variables x, y, z, functors f, g, h and predicates p, q, r. The symbol L denotes a *literal*, namely an atomic formula (i.e. a *positive* literal) or a negated atomic formula, i.e. a *negative* literal. An atomic formula is called in short an *atom*. All mentioned symbols can possibly be subscripted. The symbol θ stands for a *substitution* $\{x_1/t_1, \dots, x_n/t_n\}$, where x_i is a variable and t_i is a term for $i = 1, n$. An *application of the substitution* θ to the expression E (namely, to a term or to a formula) results in the expression $E\theta$, which is obtained from E by replacing every occurrence of the variable x_i by the term t_i. The expression $E\theta$ is called an *instance* of the expression E. Moreover, a *copy* of the expression E is an instance of E with all variables renamed to new, unique identifiers. If the given expression does not contain variables then it is called *ground*. Two expressions E and E' are *unifiable* if there exists a *unifier* for them, that is to say, a substitution θ, such that $E\theta = E'\theta$.

The semantics of FOL is given by means of an interpretation $\mathcal{I}$, which is a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consisting of an interpretation domain and an interpretation function, respectively. The formulas F and G are called *equisatisfiable* if the formula F is satisfiable whenever the formula G is satisfiable too and *vice versa*. However, the formulas F and G can have different models. Every FOL formula can be transformed to an equisatisfiable formula of the form $(Qx_1)\dots(Qx_n)F$, where Q is a quantifier (either existential or universal) and the formula F contains no quantifiers; $x_1, \dots, x_n$ are all the variables occurring in F. Furthermore, every formula can also be transformed to the *disjunctive normal form* (DNF) or to the *conjunctive normal form* (CNF). In the first case the resulting formula is a disjunction $F_1 \vee \dots \vee F_m$ where every F_i for $i = 1, m$ is a conjunction of literals. In the latter case, the result is a conjunction $G_1 \wedge \dots \wedge G_n$ where every G_i is a disjunction of literals for $i = 1, n$. Details of these transformations can be found e.g. in [3].

In our context the *reasoning* consists in proving the validity of a given hypothesis by showing that it has some particular syntactic properties. For example, the formula F is valid, namely is a tautology, if it can be transformed to the form $F' \vee \sim F'$. The proof can also be given indirectly, that is to say, by demonstrating that the negation of the formula is unsatisfiable. One of possible reasoning methods is the connection calculus [3]. A characteristic feature of this approach is that hypotheses are represented as matrices of literals. In particular, one can distinguish two forms of the representation, namely a positive and a negative one.

In the first case, a hypothesis corresponding to the given matrix is regarded as a tautology, which is to be proven directly. Every column of the matrix is considered as a conjunction of its literals and the matrix represents a disjunction of columns. All the variables are existentially quantified by assumption. In other words, the matrix stands for the formula $(\exists)H$, where the subformula H is in DNF and contains no quantifiers. The formula is said to be *positively represented* by the matrix. For example, let G denote the following formula: $\exists x \exists y(\sim p_1(x, y) \wedge p_4(a, x) \wedge \sim p_6 \vee p_2(x) \wedge \sim p_5(b) \vee \sim p_3(y))$. The matrix, which represents it positively is given on the left-hand side of Fig. 1.

In case of the negative representation one assumes that a formula corresponding to a matrix is unsatisfiable and proofs constructed for this representation are indirect. Every row of the matrix is regarded as a disjunction of its literals while the matrix corresponds to a conjunction of rows. Variables in literals are implicitly universally quantified. Summing up,

$$\begin{bmatrix} \sim p_1(x,y) & p_2(x) & \sim p_3(y) \\ p_4(a,x) & \sim p_5(b) & \\ \sim p_6 & & \end{bmatrix} \quad \begin{bmatrix} p_3(y) & & \\ \sim p_2(x) & p_5(b) & \\ p_1(x,y) & \sim p_4(a,x) & p_6 \end{bmatrix}$$

Fig. 1 Positive and negative representation of exemplary formulas

$$\begin{bmatrix} q & \\ p(a,y) & \sim q \\ \sim p(x,b) & \sim p(x,c) \end{bmatrix}$$

Fig. 2 Negative representation of the exemplary formula G_1

the matrix stands for the formula $(\forall)H$, where the quantifier-free subformula H is in CNF.

Moreover, a positive representation of any formula F can by transformed to a negative representation of the formula $\sim F$ by making an anti-clockwise quarter rotation of the corresponding matrix and negating all the literals contained in it. For example, the transformation of the matrix on the left-hand side of Fig. 1 results in the matrix on the right-hand side of the same figure. One can observe, that it negatively represents the formula $\forall x \forall y((p_1(x, y) \vee \sim p_4(a, x) \vee p_6) \wedge (\sim p_2(x) \vee p_5(b)) \wedge p_3(y))$, which is a negation of the exemplary formula G.

An easy transition between these two representations yields various advantages, which are not available in the majority of the other reasoning methods. One of them, pointed out in [3], is the possibility of a dual interpretation of a proof, i.e. either as a direct proof or as an indirect one. Another benefit of the connection calculus is the independence of a truth value of a formula, represented by a matrix, from some operations performed on matrices, which in particular correspond to commutativity of conjunction and disjunction symbols. More precisely, the following operations preserve the validity of a hypothesis given in the positive representation:

- changing order of columns in the matrix
- changing order of literals in a column
- adding a column $C\theta$ to the matrix containing the column C.

The symbol θ denotes any substitution being applied to all literals of the column C. In case of the negative representation, the unsatisfiability of the formula is an invariant of the following operations:

- changing order of rows in the matrix
- changing order of literals in the row
- adding a row $R\theta$ to the matrix containing the row R.

Let M be a matrix consisting of columns $C_1, \ldots, C_n$, which positively represents a formula F. A *path* in the matrix M is any sequence $L_1, \ldots, L_n$ of elements of M, such that the literal L_i is an element of the column C_i for $i = 1, n$. Let L be a positive literal; moreover let $neg(L) = \sim L$ and let $neg(\sim L) = L$. Two literals L and L' belonging to the same path form a *connection* if there exists a unifier θ of $neg(L)$ and L'. It should be remarked that the unifier θ has to be applied to both the columns where the literals L and L' come from. For this reason, the columns are initially copied in order to save them for future instantiations, unless they are ground.

A set of connections in the matrix M is called a *mating* for M. A mating, in turn, is called *spanning* if every possible path in M contains at least one connection included in this mating. Furthermore, a matrix is *complementary* if there exists a spanning mating for it. One of fundamental results for the connection calculus states that the formula F is a tautology if and only if M is a complementary matrix [3]. Furthermore, a necessary condition for the existence of a spanning mating for M is that M contains at least one *positive column*, i.e. a column consisting of positive literals only.

For example, one has to prove that the following formula $\forall x \forall y(q \wedge (p(a, y) \vee \sim q) \wedge (\sim p(x, b) \vee \sim p(x, c)))$, denoted by G_1, is false. The negative representation of this formula is given in Fig. 2. A transition to the positive representation of the formula $\sim G_1$, having the form $\exists x \exists y(\sim q \vee \sim p(a, y) \wedge q \vee p(x, b) \wedge p(x, c))$, results in the matrix M, which is depicted in Fig. 3.

The process of the creation of a spanning mating for M follows in four steps and results in the complementary matrix M' presented in Fig. 4. In every step one connection is built, which is indicated by a dot-ended line.

Let the symbol $L_{i,j}$ denote an element of a matrix, which is located at the i-th row and j-th column. The first step consists

$$\begin{bmatrix} p(x,b) & \sim p(a,y) & \sim q \\ p(x,c) & q & \end{bmatrix}$$

Fig. 3 Positive representation of the exemplary formula $\sim G_1$

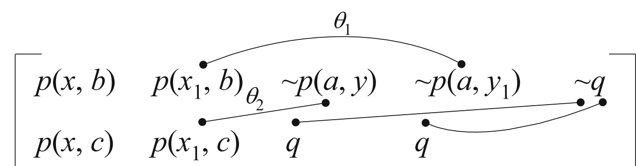

Fig. 4 A complementary matrix for the exemplary formula $\sim G_1$

in forming a connection between literals $L_{1,1}$ and $L_{1,2}$ from the matrix M. For this purpose, the columns C_1 and C_2 from M are copied into the columns C_2 and C_4 in the matrix M', respectively. In the next step, a unifier $\theta_1 = \{x_1/a, y_1/b\}$ is applied to them and the connection is formed. Subsequently, the literal $L_{2,2}$ is connected to the literal $L_{1,3}$ in the matrix M' using the substitution $\theta_2 = \{y/c\}$. It should be noticed that the column C_2 is not copied since it becomes ground in the former step. Also, for the sake of simplicity, a copy of the column C_3 is neglected as unnecessary for the creation of a spanning mating. In two final steps, the literals $L_{2,3}$ and $L_{2,4}$ are respectively connected to the literal $L_{1,5}$.

3 Reasoning method

The main goal of the reasoning algorithm is the creation of the spanning mating for the given matrix M. To do this, one has to check if every possible path in the matrix contains at least one connection. Let P denote a path being currently under construction. In other words, the matrix contains some columns, which the path P has not been conducted through. Hence, one has to select a column C of this type and then select a literal L from it. If P contains a literal L', which forms a connection with the literal L, then the creation of the path going through L can stop since every path starting from it contains a connection. Otherwise, the literal L should be added to P. The act of connecting a literal L to some element of the path P is called a *reduction step*, while adding a new literal to the path is an *extension step* [8].

As said in Sect. 2, a copy of every nonground column participating in a reduction step is added to the matrix M. Due to undecidability of FOL, this can result in an unlimited growth of a number of columns, which in turn prevents from finding a suitable mating for M. However, in some of such cases the spanning mating could be found if columns and literals were selected in a different order. This problem can be overcome by means of the *depth-first iterative deepening* (DFID) search strategy. Roughly speaking, the process of constructing a spanning mating for a given input matrix starts with a limited length of a path. Therefore, all possible paths are tried, whose length does not exceed the limit. If no spanning mating is found, the process starts again with a length limit increased. The initial limit, as well as its increment in every iteration are parameters of the DFID search. This method is proven to be asymptotically optimal among brute-force strategies in terms of proof length, space and time [4]. Hence, it is widely used in lean reasoners for undecidable logics.

Below, we present a reasoning algorithm for the connection calculus, whose general description is to be found in [3]. The algorithm is given in pseudocode since we try to keep as close as possible to the realization presented in the next section. Furthermore, the algorithm has a form of two subprograms, namely the function $Prove1$ and the procedure $Prove2$. The subprograms, in turn, consist of sequences of steps or statements, which are executed one by one. Each of them can either terminate (implicitly) successfully or with a failure. In the latter case, the whole subprogram terminates with a failure. Otherwise, namely if all elements of the subprogram terminate successfully then the subprogram terminates successfully, as well.

The pseudocode contains some boldfaced statements. In their description, we use the following metasymbols: $Expr$ (an expression), $Cond$ (a boolean expression), $Stat$ (a statement or a sequence of statements), Set (a set or a sequence). Moreover, the metasymbol $Elem$ stands for the specification of an element, which is to be selected from a given set. The specification comprises an identifier of the element. It may also contain some conditions and constraints, which have to be satisfied for a successful selection. All the mentioned symbols can possibly be subscripted. Additionally, the symbol $[Expr]$ denotes an optional expression $Expr$.

The statement **if** $Cond_1$ **then** $Stat_1$ [**elseif** $Cond_2$ **then** $Stat_2$] [**else** $Stat_3$] represents a decision. It has an intuitive semantics, whose definition is to be found e.g. in [9].

The statement **select** $Elem$ **from** Set declares a nondeterministic choice of the element $Elem$ from the set Set. In other words, any element can be selected, which satisfies the $Elem$ description. This issue is discussed in the sequel. After a successful selection of the element $Elem$, it is assumed to be removed from Set for further processing. On the other hand, the statement results in failure if the selection of an element is impossible, because for example Set is empty or it contains no element specified by $Elem$.

The statement **for** $Elem$ **in** Set **do** $Stat$ represents an iterative selection of the element $Elem$ from the set Set. Unlike the former statement, it always assumes an order of the given set. After a subsequent element is selected, it is considered to be removed from Set for the current iteration, which consists in the execution of the statement (or a sequence of statements) $Stat$. Furthermore, if $Stat$ executes the statement **continue** then the current iteration is interrupted and the control is passed to the next one.

The statement **exit** causes a successful termination of a procedure. The statement **return** $Expr$ acts similarly when used in a function, except it returns the value of $Expr$ to the function call. On the other hand, the statement **fail** makes a subprogram stop with a failure.

A top-level part of the algorithm is the function $Prove1$. It takes the matrix M, represented as a set of columns, and tries to build a spanning mating for it with respect to the limit Lim. One should observe, that the function returns the value *true* only in case of a successful termination of the step 3.

Function $Prove1(M, Lim)$
Input: M – a matrix, Lim – a limit of a path length.
Output: the answer $true$ if M is complementary; the answer $false$ if M is not complementary; a *failure* if no spanning mating can be constructed for M with respect to Lim.
Step 1: **if** M contains no positive column **then return** *false*.
Step 2: **select** a positive column C **from** M.
Step 3: execute $Prove2(C, M, \varnothing, Lim)$.
Step 4: **return** *true*.
End Function.

The procedure $Prove2$ initially checks if the current column C, regarded as a set of literals, is empty. Such a column corresponds to a true formula (as an empty conjunction) and thus the matrix containing it is a tautology, as well. Otherwise, for every element L of the column C the reduction step (i.e. step 3) is tried. If the reduction is not possible, the extension step is undertaken, which is represented in the procedure by steps 3–8.

In particular, step 5 consists in the creation of a copy C'' of the column C'. However, making a copy of a column and reusing it in subsequent computations is pointless if the column is ground. In such a case, the column C' is not considered for further processing (see step 8). Moreover, the symbol $L|P$ denotes a path constructed from the path P by adding L to it as the first element; the value of the expression $length(P)$, in turn, is a number of elements of the sequence P. The function *unify*, given in step 7, realizes a unification algorithm. It results in failure if the arguments are not unifiable.

Procedure $Prove2(C, M, P, Lim)$
Input: C – a current column, M – a set of columns that the current path is to be conducted through, P – a current path, Lim – a limit of a path length.
Output: *success* if each path starting from P and leading through every element of C, and then through all columns in M, contains a connection; a *failure* otherwise.
Step 1: **if** $C = \varnothing$ **then exit.**
Step 2: **for** every literal L **in** C **do** execute the steps 3–8.
Step 3: **if** there exists a literal L' in P and a unifier θ of $neg(L)$ and L' **then** $P := P\theta$, $C := C\theta$, **continue.**
Step 4: **select** a column C' **from** M.
Step 5: $C'' := copy(C')$.
Step 6: **select** a literal L' **from** C''.
Step 7: $\sigma := unify(L', neg(L))$.
Step 8: **if** C' is ground **then** execute $Prove2(C''\sigma, M, (L|P)\sigma, Lim)$ **elseif** $lenght(P) < Lim$ **then** execute $Prove2(C''\sigma, M \cup \{C'\}, (L|P)\sigma, Lim)$ **else fail**.
End Procedure.

4 System description

Every reasoning system contains two basic elements. One of them is a realization of inference rules defining conclusions that can be obtained from premises. The other element is a strategy of searching for a proof, which determines a way the premises are selected in a reasoning process. Both the elements are naturally present in logic programming languages, e.g. in Prolog. Hence, reasoning systems are often implemented as logic programs. Unfortunately, the majority of the considered languages handle only one search strategy being fixed in the execution environment.

One of the exceptions is Oz—an experimental, multiparadigm programming language [9], whose execution environment is the Mozart platform [13]. The work on the language, as well as on its software platform began in nineties by a group of European laboratories, whose significant participants were German Research Centre for Artificial Intelligence (DFKI), Swedish Institute of Computer Science, Universite catholiqué de Louvain (Belguim) and Universität des Saarlandes (Germany). The Oz language enables the usage of many well-known programming models together in the same program, e.g. imperative and declarative programming, distributed programming, etc. Every model, called a *programming paradigm*, is represented by a characteristic set of Oz instructions. The paradigm corresponding to logic programming is named a *relational paradigm*. A declarative semantics of a relational program is similar to the Prolog one. However, a search strategy is not fixed like in the case of Prolog, but it is specified as a parameter of the program execution. Therefore, the same program can be run according to various search strategies. This possibility is very convenient, especially in the prototyping and testing phase. A realization of a search strategy has a form of a special object called a *search engine* (in short: an engine). A number of engines are available in libraries of the Mozart platform. They are implemented in the Oz language at an abstract level. In consequence, definitions of engines are relatively short, so they can be modified and extended with no particular effort. All these reasons decided that the system presented in the paper is realized as a relational program in the Oz language.

An operational semantics of a relational program is described using a *search tree*, whose nodes correspond to *computation spaces* (in short: spaces) [10]. The main use of spaces is to encapsulate computations. In other words, computations running in spaces are separated one from another and thus they can be performed independently. Going into some details, an execution of a program starts in the root space. Let us assume that the program contains a statement, which introduces a nondeterministic choice with n alternatives. In such a case, the respective node of the search tree has n child nodes. Each of them is a clone of the parent

one, except it contains the information, which alternative it represents. Moreover, the execution of subsequent "nondeterministic statements" results in the further branching of the tree. Every dangling node corresponds to one possible result of computations, which can either be a success or a failure. Summing up, a relational program fully determines the shape of its search tree. The tree, however, is not built by the program but by a search engine. In this way the declarative semantics of a program, corresponding to the structure of the tree, is separated from its operational semantics. It should be underlined that computations performed in every branch are independent from the other branches. More precisely, they compete among one another for finding a solution. In consequence, a search tree can be easily constructed in parallel on distributed machines.

An appropriate, distributed search engine is an instances of the library class `Search.parallel`. It can be regarded as a team of concurrent autonomous agents comprising a *manager* and a group of *workers*. The manager controls the computations by finding a work for idle workers and collecting the results whereas the workers construct fragments of the search tree. Members of the team communicate by exchanging messages. A command, which creates a new engine specifies a computational environment, namely a set of machines connected in a network. The command also indicates a number of workers to be run on each of the machines. The engine starts on behalf of the manager, which sends a root of the search tree to a selected worker and puts the worker on a list of possibly busy workers. On the other hand, every idle worker sends a request for a job to the manager. In response, the manager tries to find a busy worker (registered on the list), which is ready to share its job. If such a worker is found, it sends a root of an unexplored subtree to the manager, which conveys it to the idle worker and puts this worker on the list. A worker can also inform the manager about finding a solution. After receiving such a message, the manager may tell all the workers to stop their jobs and close the engine. The detailed description of the engine architecture, including the communication protocol, is to be found in [10].

The reasoning system processes matrices, which are represented by Oz data structures. In particular, a matrix is a list of columns, while a column is a list of literals. We use the symbol O in the superscript to denote the Oz representation of the given expression. A negative literal has a general form of Oz *record* ([9]) `neg(`L^O`)`, whose *label* is a reserved symbol `neg` standing for the negation and the *field* L^O is an Oz counterpart of the positive literal L. In other words, L is an atom, namely a predicate possibly followed by a tuple of terms. Predicates, functors and constants are denoted by alphanumeric strings starting from a lowercase letter, whereas variable names start from an uppercase one. A tuple of terms $(t_1, \ldots, t_n)$ is represented by the expression `(`t_1^O t_2^O `...` t_n^O`)` with a space character as a separator. Summing up, the exemplary matrix given in Fig. 1 has the following representation in Oz: `[[neg(p1(X Y)) p4(a X) neg(p6)] [p2(X) neg(p5(b))] [neg(p3(Y))]]`.

The key part of the system is the function `Prove1` and the procedure `Prove2`, which are defined below. They are a straightforward realization of the reasoning method described in Sect. 3. A crucial element of this realization is the selection mechanism, which corresponds to the statement **select** *Elem* **from** *Set*. It is used for choosing both a column from the matrix and a literal from the column. The mechanism has a form of the statement `{SelectNth Lst {Space.choose {Length Lst}} Elem Lst1}`. The procedure `SelectNth` selects the n-th element `Elem` from the list `Lst`, where n is a number specified by the second argument. The argument `Lst1` is a list obtained by removing the element `Elem` from `Lst`. The function call `{Length Lst}` returns a number of elements of the list `Lst`. The statement `{Space.choose N}` is executed in a computation space and it informs a search engine about a nondeterministic choice with `N` alternatives. The engine in the response clones the space `N` times and sends to each copy a numerical identifier ranging from 1 to `N`. The identifier becomes a value of the expression `{Space.choose N}`. In consequence, each element of the list `Lst` is to be processed further in a separate space.

The first argument (`Mat`) of the function `Prove1` is an input matrix, while the latter one (`Lim`) is a current limit of the path length. Generally, the call of this function may yield three different results. In particular, the value `true` is returned if one can construct a spanning mating for the matrix `Mat` and thus the formula represented by `Mat` is a tautology. On the other hand, i.e. when it can be proven that the spanning mating for `Mat` does not exist, the value `false` is returned. The function can also terminate by failure, which means that no spanning mating can be constructed for `Mat` with respect to the current limit of the path length. In such case, the computations can be repeated in a DFID manner, namely with an increased value of the argument `Lim`, as it is explained in Sect. 3.

The function initially checks if the matrix `Mat` is empty (line 2). If so, it returns `false` (line 7). Otherwise, a separate clone of the current space is created for each column `C` of the matrix `Mat` (line 3). If the column is not positive, i.e. it contains at least one negative literal, the further computations cease in failure (line 4); the function call `{Record.label L}` returns a label of the record `L`. In other case, that is to say, if the column `C` is positive, the procedure `Prove2` is executed for it (line 5).

```
1  fun {Prove1 Mat Lim}
2     if Mat \= nil then C Cs in
3        {SelectNth Mat {Space.choose {Length Mat}} C Cs}
4        if {Some C fun {$ L} {Record.label L} == neg end} then fail else
5           {Prove2 ['!'] (neg('!')|C)|Cs nil Lim}
6           true end
7     else fail end
8  end
```

The execution of the procedure `Prove2` results in failure if no spanning mating can be constructed for the input matrix with respect to `Lim`. In consequence, it causes a failure of the function `Prove1` (in line 5). On the other hand, namely when the mating is constructed, the procedure `Prove2` terminates successfully and the function `Prove1` returns the value `true`.

The first argument of the procedure `Prove2` (i.e. `Col`) is a list of literals corresponding to the current column. The argument `Mat`, in turn, is a list of remaining columns, which the current path is to be conducted through. The path, having a form of a list of literals, is represented by the argument `Path`. The last argument (`Lim`) is a number limiting the `Path` length. The initial call of the procedure (line 5 of the definition of the function `Prove1`) contains a technical trick, used at first in the system leanCoP. It consists in building an artificial current column `['!']`, which encompasses a reserved predicate `'!'`. This symbol must not occur in the input matrix. Furthermore, the original current column `C` is extended by the negation of the predicate `'!'` and added to the list of remaining columns, which is represented by the expression `(neg('!')|C)|Cs`. The aim of this artifice is to cause the initial current column `C` to be copied (line 11 of the definition of the procedure `Prove2`). Otherwise, the reasoning system is incomplete, namely it may not be able to prove some formulas, which are tautologies.

```
1  proc {Prove2 Col Mat Path Lim}
2     if Col \= nil then Col1 Col2 L Ls NegL in
3        L = Col.1 Ls = Col.2
4        if {Record.label L} == neg then NegL = L.1 else NegL = neg(L) end
5        if {UnifiableMember NegL Path} then
6           {Prove2 Ls Mat Path Lim}
7        else
8           if Mat \= nil then Mat1 in
9              {SelectNth Mat {Space.choose {Length Mat}} Col1 Mat1}
10             if Col1 \= nil then L1 Ls1 in
11                {CopyTerm Col1 Col2}
12                {SelectNth Col2 {Space.choose {Length Col2}} L1 Ls1}
13                if {Not {OccurUnify L1 NegL}} then fail end
14                if {Not {NotGround Col1}} then
15                   {Prove2 Ls1 Mat1 L|Path Lim}
16                elseif Lim > {Length Path} then
17                   {Prove2 Ls1 Mat L|Path Lim}
18                else fail end
19             else fail end
20          else fail end
21          {Prove2 Ls Mat Path Lim} end
22    else skip end
23 end
```

Initially, the procedure `Prove2` checks if the current column `Col` is empty (line 2). If so, the computations cease successfully (line 22) since the input matrix represents a true formula. In other case, namely when the current column is not empty, it is split to the current literal `L` and the remaining literals `Ls` (line 3). Then, the negated literal `NegL` is constructed for `L` (line 4) and the reduction step is tried for it (line 5). More precisely, the literal `L` forms a connection if `NegL` is successfully unified with some element of the current path. This ends the path led through the current literal. In consequence, the process of creation a spanning mating is continued for remaining literals of the current column (line 6). Otherwise, to wit when no reduction can be performed for `L`, the extension step is applied in lines 8–21.

It starts from checking if the list of remaining columns (i.e. `Mat`) is empty (line 8). In such case, any extension is impossible and thus the procedure terminates by failure (line 20). Otherwise, a column `Col1` is selected from `Mat` (line 9). If the column is empty (line 10) then it can not be used for an extension step and therefore the computations result in failure for it (line 19). In other case, the procedure creates a copy of `Col1` (line 11). It should be reminded that all variables appearing in `Col1` are replaced in the copy `Col2` by new, unique ones. The next step consists in the selection of a literal `L1` from the column `Col2`. If the literal does not form a connection with the current literal `L` then the computations terminate by failure for it (line 13). Otherwise, the current path can possibly be extended by the literal `L`, nevertheless, with regard to some other conditions. In particular, when the column `Col1` is ground (line 14), that is to say, it contains no variables, then it can be removed from the list of remaining columns for further processing. Such a

list is indicated by `Mat1`. At last, the literal `L` is added to `Path` and the computations are continued for the other literals `Ls1` in the column `Col1` and for the list `Mat1` (Line 15). It should be remarked, that in the considered case the column `Col1` is identical to its copy `Col2`. On the other hand, when the column `Col1` is not ground, it can not be neglected in further processing. Therefore, the computations are continued for literals `Ls1` and for remaining columns `Mat` with `Col1` included (line 17). However, in this case the length of the current path must not exceed the limit `Lim` (line 16). Otherwise, the computations result in failure. Finally, after successful processing of the current literal `L`, the computations are continued for the rest of the current column, i.e. for the remaining literals represented by the symbol `Ls` (line 21).

Table 1 The computational time [s] for selected TPTP problems

Problem	1 Worker	2 Workers	3 Workers	4 Workers	5 Workers
SYN212-1	101.88	77.37	56.86	46.01	40.12
SET052-6	109.56	26.03	9.63	7.18	6.71
PUZ037-2	151.08	80.64	56.16	37.2	33.67
PLA010-1	190.41	167.78	146.44	112.19	61.55
MGT030-1	214.29	80.83	61.59	52.41	50.88
LCL009-1	59.68	56.8	56.23	5.79	5.18
GEO026-3	252.73	148.9	112.15	78.07	63.21
BOO012-1	33.34	33.11	5.28	5.29	5.28
ANA029-2	240.6	19.91	17.29	11.34	7.07

5 Experimental results

The experiments discussed in this section were aimed at estimating the speedup obtained by running the reasoning system in parallel on distributed machines with the increasing number of workers. The computational environment consisted of identical machines equipped with the processor Pentium P4D 3.4 GHz, 1 GB RAM, 1GBit Ethernet and Linux 2.6.32. All the computers were powered by the Mozart system 1.3.2. One of the machines was designated for running the manager while each of the other computers processed one worker.

All the testing hypotheses are tautologies coming from the TPTP library [12]. The formulas were chosen under the general criterion that the time of computations performed by one worker should be in the range from 30 to 300 s. Tests were run with the initial input depth equals 1 which was successively increased by 1. The results of the tests are collected in Table 1. The subsequent header row contains a problem name and a number of workers appearing in the given variant of the environment. Every entry in the table represents a computational time taken from the system clock as an arithmetic mean of 5 runs.

A speedup of computations is depicted in the bar chart given in Fig. 5. All bars are grouped in clusters corresponding to testing problems. Let T_i denote the time of computations performed in the environment consisting of the manager and i workers. Every bar represents a speedup $S_i = T_1/T_i$ obtained for the given problem and for $i = 2, 5$. The value of i is indicated by a shading of a bar, as it is given in the legend on the right side of the chart.

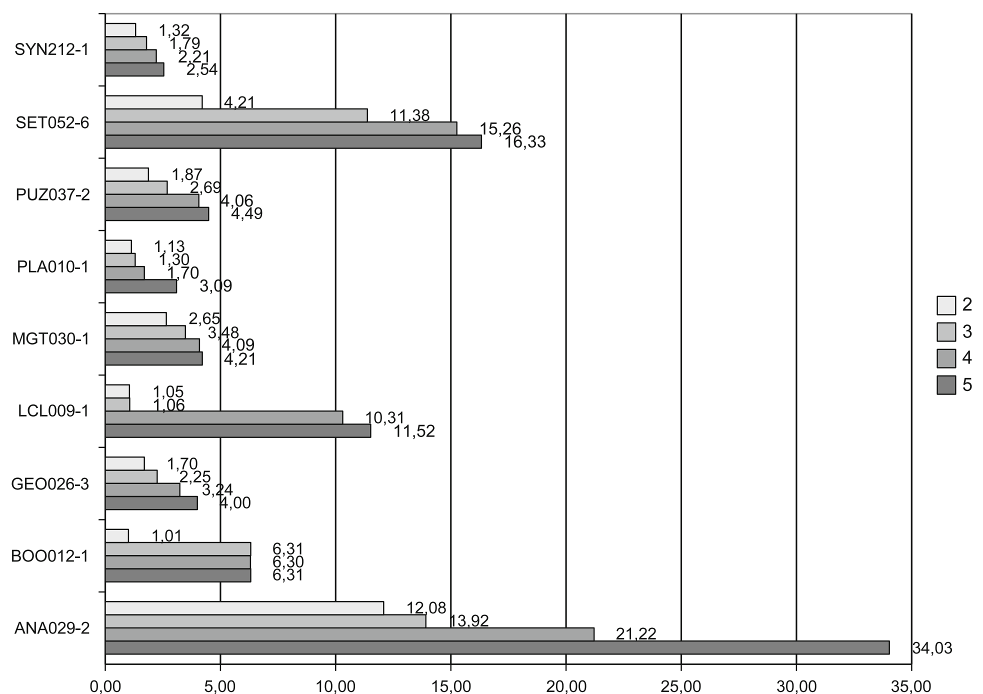

Fig. 5 Speedup for testing problems

Distinct speedup characteristics obtained for the input problems confirm rather an expected effect that the speedup depends on the structure of the constructed search tree. An increment of the speedup is nearly always positive except one case, i.e. the problem BOO012-1. However, the speedup fluctuates with the increasing number of workers. In some cases one can observe a surge in speedup, e.g. for problems SET052-6 (from S_2 to S_3), LCL009-1 (from S_3 to S_4), BOO012-1 (from S_1 to S_2) or ANA029-2 (from S_3 to S_4). It can be explained by the fact that the engine constructs the search tree until the first successful branch is found. Thus, the results of experiments strongly depend on the way the search tree is partitioned among the workers. In particular, the computational time reduces drastically if the failed branch appears as the first one in the subtree given to some worker. Another observation is the computational time nearly stops to decrease after reaching a value of around 5 s., e.g. for problems LCL009-1 or BOO012-1. This limitation may follow from the time overhead imposed by communication mechanisms of the network environment.

6 Final remarks

In this paper we describe a simple, distributed reasoning system for the first order logic. The system fits the lean deduction style and it uses a connection calculus as a reasoning method. The system is actually independent from the computational strategy thanks to implementing it in the relational model in the Oz language. We use the parallel search engine from the Mozart programming environment to run the system on distributed machines. Experiments show a reasonable speedup achieved by distributing the computations for some exemplary problems. However, the comprehensive analysis of the system efficiency requires more tests. The tests have to consider a greater number of workers and also a greater number of problems selected with regard to the structure of the search tree. These tests are planned for the future.

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

1. Amir, E., Maynard-Zhang, P.: Logic-based subsumption architecture. Artif. Intell. **153**, 167–237 (2004)
2. Beckert, B., Possega, J.: leanT^AP: Lean, Tableau-based deduction. J. Autom. Reason. **15**(3), 339–358 (1995)
3. Bibel, W.: Automated Theorem Proving. Vieweg Verlag, Braunschweig (1987)
4. Korf, R.E.: Depth-first iterative-deepening: an optimal admissible tree search. Artif. Intell. **27**, 97–109 (1985)
5. Lloyd, W.L.: Foundations of Logic Programming, 2nd edn. Springer, Berlin (1987)
6. Manthey, R., Bry, F.: SATCHMO: A Theorem Prover Implemented in Prolog. LNCS, vol. 310, pp. 415–434. Springer, Berlin (1988)
7. Neugebauer, G., Schaub, T.: A Pool-Based Connection Calculus. Technical Report AIDA-91-02, TH Darmstadt (1991)
8. Otten, J., Bibel, W.: leanCoP: lean connection-based theorem proving. J. Symb. Comput. **36**(1–2), 139–161 (2003)
9. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. The MIT Press, Cambridge (2004)
10. Schulte, Ch.: Programming Constraint Services. LNCS, vol. 2302. Springer, Berlin (2002)
11. Stickel, M.: A Prolog Technology Theorem Prover: A New Exposition and Implementation in Prolog. Technical Note No. 464, SRI Int., Manlo Park (1989)
12. Sutcliffe, G., Suttner, C.B., Yemenis, T.: The TPTP Problem Library. LNCS, vol. 814, pp. 252–266. Springer, Berlin (1994)
13. The Mozart Programming System. http://www.mozart-oz.org. Accessed 29 Nov 2013